

EV368629676

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

APPLICATION FOR LETTERS PATENT

**Evaluating Queries Against In-Memory Objects
Without Serialization**

Inventors:

Umesh Madan

Geary L. Eppley

David Wortendyke

ATTORNEY'S DOCKET NO. MS1-1824US

TECHNICAL FIELD

The systems and methods described herein generally relate to filtering input for data processing using filters and, more particularly, to systems and methods for evaluating an object of one type against a filter of another type.

BACKGROUND

Computing systems - i.e. devices capable of processing electronic data such as computers, telephones, Personal Digital Assistants (PDA), etc. - communicate with other computing systems by exchanging data according to a communications protocol that is recognizable by the systems. A system utilizes filters to analyze data that is sent and/or received by the system and to determine if and how the data will be processed further.

Filters are a set of one or more queries against which an input is tested. For purposes of the present discussion, the terms "filter" and "query" are singular and interchangeable, and multiple queries are said to comprise a "filter table." One way filters are used is to allow an input into a system only if the input satisfies one or more particular filters. Another way filters are used is for message processing, such as with a mail program. Inputs that satisfy a filter associated with a particular mailbox are forwarded to the mailbox. Filters can also be used for file access algorithms, security controls, etc.

Filters are written in a suitable query language such as XPath or XQuery. Query languages are designed to operate on data structured according to a particular format. For example, XPath and XQuery are designed to operate on data structured as XML (eXtensible Markup Language). But sometimes, a filter designed to operate on data structured according to one format are called on to

1 evaluate data structured according to another format. In such instances, a
2 translation process must be performed so that the object can be tested against the
3 filter. However, data transform procedures can consume significant resources.

4 Once such instance occurs when an XPath query is called on to evaluate a
5 CLR (Common Language Runtime) or Java object. A typical way to perform this
6 evaluation is to serialize the object and then use the serialized data to build a
7 template of the object according to XML. The XPath query can then be evaluated
8 against the object using the XML structure.

9 But the serialization process is expensive because it involves maintaining
10 buffers, writing strings, etc. The resultant XML must then be parsed to build
11 (typically) a DOM. Furthermore, an object is often at the root of a tree/graph of
12 objects and can have references to other objects. Therefore, serializing an object
13 can sometimes involve serializing multiple objects which adds to the overhead
14 required for processing.

15 **SUMMARY**

16 At least one implementation described herein relates to evaluating queries
17 (i.e. filters) constructed according to a query language (e.g. XPath) against an
18 object constructed according to a non-conforming format (e.g. a CLR object)
19 without serializing the object. An info set (information set) model that conforms to
20 a query language format (e.g. XML for XPath) is derived that maps object fields
21 and properties to info set information items. An information item is identified and
22 a corresponding property is located in the CLR object. The data related to the
23 corresponding CLR object property is retrieved and compared to the query value
24 to determine if the values match. No serialization is required in this process.

1 An infoaset model is only developed to the extent necessary to locate a
2 property that corresponds to an information item being tested. Once the object
3 value is determined and tested, further development of the infoaset model is
4 unnecessary. The portion of the infoaset model that has been developed is stored so
5 that subsequent queries against the same object can utilize the same infoaset model
6 and augment it if further development of the model is required.

7 In at least one implementation, opcodes are generated that, when executed,
8 evaluate a query. The opcodes dynamically generate 'helper' code that performs a
9 function that is the same as what would be performed by intermediate language
10 (IL) instructions compiled from source code. Generating code implies emitting IL
11 instructions into dynamic assemblies. This generated IL is just in time (JIT)
12 compiled in a .NET runtime and allows the opcodes to retrieve property values
13 and fields directly from CLR objects as though the opcodes were executing
14 compiled hand-written source code.

15 **BRIEF DESCRIPTION OF THE DRAWINGS**

17 A more complete understanding of exemplary systems and methods
18 described herein may be had by reference to the following detailed description
19 when taken in conjunction with the accompanying drawings wherein:

20 Fig. 1a is a representation of an object declaration.

21 Fig. 1b is an instance of an object according to the object declaration of Fig
22 1a.

23 Fig. 2 is a representation of an XML construct corresponding to the object
24 of Fig. 1b.

25 Fig. 3a is a representation of an infoaset model for a CLR object.

1 Fig. 3b is a representation of an XML template corresponding to the info set
2 model depicted in Fig. 3a.

3 Fig. 4 is a block diagram of a system in accordance with the description
4 provided herein.

5 Fig. 5 is a flow diagram that depicts a methodological implementation
6 evaluating a query against a non-conforming object.

7 Fig. 6 is a diagram of an exemplary computing environment in which the
8 implementations described herein may operate.

9 **DETAILED DESCRIPTION**

10 The present disclosure relates to evaluating a query (i.e. filter) against a
11 non-conforming object. As used herein, the term “non-conforming” means that
12 the object is structured according to a format that is different than a format to
13 which the query conforms. For example, the following description relates
14 specifically to evaluating an XPath query against a CLR (Common Language
15 Runtime) object. It is noted, however, that the subject matter is not necessarily
16 limited to XPath queries or CLR objects. The techniques described herein may be
17 used with other types of queries that are evaluated against objects that do not
18 conform to the query format.

19 A messaging system is one type of system that utilizes filters or queries to
20 process inputs. Messaging systems utilize message handlers (i.e. message routers)
21 that refer to filter tables to make processing decisions about particular messages.
22 Filters stored in the filter table test a set of conditions, or rules, against message
23 content and return a value of true if the conditions are satisfied.
24
25

1 Message handlers typically bind a filter to an action that is executed when
2 the filter is satisfied. The logic of such binding usually takes the form of “If
3 message matches filter X, take action Y” (action Y is associated with filter X).
4 The logic may also represent “Test a set of filters, S, against a message and select
5 a subset, S1, of filters that match the message; then take one or more actions
6 associated with the filters of S1.”

7 Messaging systems modeled on XML utilize filters defined using an XML
8 query language, such as XPath. However, such systems often deal with messages
9 that are non-XML based, such as messages and/or message headers that are
10 strongly-typed in-memory CLR objects, which are designed primarily for efficient
11 programmatic access.

12 A typical way to run an XPath query over a CLR object is to serialize the
13 object (i.e. message objects) and convert the serialized data to an XML format. An
14 alternative method could maintain a parallel XML version of every message
15 header and property that is used solely for query processing, including headers that
16 are never transmitted. Unfortunately, either of these methods imposes a significant
17 burden on computational resources.

18 A substantially more efficient method and system for performing such
19 methods are described herein. In the described systems and methods, an XPath
20 query is evaluated against in-memory CLR objects directly, without requiring
21 serialization of the CLR object. Such techniques result in performance gains of
22 several orders of magnitude over the other techniques referenced above.

Exemplary Object Structure

Fig. 1a depicts an exemplary object structure 100 (expressed here by a class definition for a “book” object) that could be used to represent information regarding one or more book objects. The object structure 100 includes a root class 102 (“public class book”). The root class 102 includes a “Title” property 104, an “Author” property 106 and a “chapter” array property 108 (“chapters.”)

The exemplary object structure 100 also includes a class 110 (“public class chapters”) that has a “chapternum” property 112 (identifying a chapter number), a “chapter title” property 114, and a “text” property 116. It is noted that the object structure 100 could include several other properties that are not shown.

It is noted that the exemplary object 100 takes a hierarchical form just as XML data is arranged hierarchically. The fact that both types of data are hierarchical provides a natural and convenient way to relate an object to an XML construct.

Fig. 1b is an instance of an object 120 according to the object structure 100 shown in Fig. 1a. The object 120 is populated according to the novel “Moby Dick” and, therefore, has a title property 122 with a value of “Moby Dick.” The object 120 also includes an author property 124 having a value of “Melville.”

The object also includes two instances of the public class “chapters” (128, 136). The first instance 128 has a chapternum property 130 having a value of “1” that corresponds to chapter one of the novel. The first instance 128 also has a chapter title property 132 with a value of “Loomings” and a text property that includes the contents of the first chapter (beginning with “Call me Ishmael.”).

The second instance 136 has a chapternum property 138 having a value of “2”, a chapter title property 140 with a value of “The Carpet-Bag”, and a text

1 property 142 that includes the contents of the second chapter of the novel (which
2 begins with “I stuffed a shirt or two....”). Other instances of the public class
3 “chapters” may be included but are not shown here.

4 The object 120 is a typical non-XML object that may be encountered in
5 query processing. Further examples will refer to the exemplary object 120 to
6 further demonstrate the techniques shown and described herein.

7 Exemplary XML Construct

8 Fig. 2 depicts an exemplary XML construct 200 that corresponds to the
9 exemplary object 120 shown in Fig. 1b. In other words, if a similar instance of the
10 object 120 were written in XML, the result would appear as shown in Fig. 2. The
11 XML construct 200 is shown populated with data from “Moby Dick” similar to
12 Fig. 1b. For convenience, only a portion of the book instance is shown.

13 The XML construct 200 includes a root element (“book”) 202. The root
14 element 202 has a first child node 204 (“Title”) having a value of “Moby Dick”, a
15 second child node 206 (“Author”) having a value of “Melville” and a third child
16 node 208 (“chapters”), referred to below as a “chapters node” 208.

17 The chapters node 208 has a first chapter node 210 and a second chapter
18 node 212. Each chapter node 210, 212 includes three child nodes: a chapter
19 number node 214, 220; a chapter title node 216, 222; and a text node 218, 224.
20 (The nodes are also known as XML elements, but are referred to herein as nodes to
21 better allude to the hierarchical nature of XML.)

22 The chapter number node 214 of the first chapter node 210 has a value of
23 “1”. The chapter title node 216 of the first chapter node 210 has a value of
24 “Loomings”, and the text node 218 of the first chapter node 210 has a value
25 beginning with “Call me Ishmael.”

1 The second chapter node 212 includes a chapter number node 220 having a
2 value of “2”, a chapter title node 222 with a value of “The Carpet-Bag” and a text
3 node having a value beginning with “I stuffed a shirt or two...”

4 The similarity between the hierarchies of the XML construct 200 and the
5 object 120 (Fig. 1b) are apparent. It is this hierarchical nature of the construct 200
6 and the object 120 that are exploited in the systems and methods described herein
7 to allow elements and values to be located in a non-XML object without first
8 having to serialize the object into an XML format. Since the XML format that
9 would result from serialization can be determined prior to serialization, the need to
10 serialize the non-XML object is rendered unnecessary.

11 Exemplary Infoset

12 XPath expression target XML Infosets whereas CLR objects have no notion
13 of XML constructs such as elements and attributes. To execute XML queries
14 directly on CLR objects, the CLR objects must be interpreted as XML. To do so,
15 the same heuristics that are used to serialize CLR objects can be applied to
16 interpret the CLR objects as XML, but without serialization.

17 An infoset model for a CLR-Type is a mapping of the CLR-Type properties
18 to XML. An infoset model for a CLR-Type provides a fixed template for its
19 corresponding XML. Although the XML text nodes will vary for every instance
20 of the Type, the XML markup - element tags, attribute names and namespaces -
21 remain unchanged.

1 **Fig. 3a** depicts an infoaset model 300 for the following exemplary CLR-
2 Type:

```
3       public class Envelope  
4       {  
5             public ActionHeaderObj ActionHeader {get'}  
6             public string Version {get;}  
7       }  
8  
9       public class ActionHeaderObj  
10      {  
11          {XmlElement ("Action")}  
12          public string ActionUri {get;}  
13  
14          {XmlElement ("HeaderName")}  
15          public string Name {get;}  
16  
17          {XmlAttribute}  
18          public string ID {get;}  
19      }  
20  
21  
22  
23  
24  
25
```

12 The infoaset model 300 includes a mapping for every public property that is
13 serializable into XML. The infoaset model 300 is shown using pseudo syntax. An
14 actual infoaset model may appear differently, but the structure is similar.

15 The infoaset model 300 includes an envelope class 302 that has the
16 following children 304: an ActionHeader property 306 and a Version property 308.
17 The infoaset model 300 also includes an ActionHeaderObj class 310 that has the
18 following children 312: an Action property 314 and a HeaderName property 316.
19 The Action property 314 in the present example denotes an ActionURI that is the
20 object of the Action property 314. The ActionHeaderObj class 310 also includes
21 an attribute 318, namely, a property ID 320.

22 The infoaset model 300 is created at runtime for at least two reasons. First,
23 filters are created and added to filter tables at runtime. Second, a given filter is
24 applicable to and satisfied by any object that has a matching XML representation.
25 The full domain of objects that a query could test is undeterminable in advance.

1 **Fig. 3b** depicts an XML template 330 that corresponds to the infoaset model
2 300 shown in Fig. 3a. Based on the infoaset model 300, instances of the Envelope
3 object would be similar to the XML template 330 if they were structured as XML.

4 The XML template 330 includes a root element 332 (“Envelope”) that
5 corresponds with the envelope class 302 of the infoaset model 300. The root
6 element 332 includes two child nodes: an ActionHeader element 334 and a Version
7 element 340. The ActionHeader element 304 has a value 342 that corresponds
8 with the property ID 320 of the infoaset model 300.

9 The ActionHeader element 334 also includes two child nodes: an Action
10 element 336 and a HeaderName element 338. The Action element 336
11 corresponds with the Action property 314 of the infoaset model 300 and the
12 HeaderName element 338 corresponds with the HeaderName property 316 of the
13 infoaset model 300.

14 An infoaset model may be constructed in its entirety when a new object is
15 encountered or only to an extent necessary to determine a value in the object
16 specified by a query. In the following description, the latter implementation is
17 described. It is more efficient to construct only the part of the infoaset model that is
18 needed because the remainder of the infoaset model may never be required.

19 Any portion of an infoaset model that is constructed is stored so that when a
20 corresponding object is subsequently encountered, the work already performed is
21 not repeated. If an object has been previously encountered and a partial infoaset
22 has been built, then the partial infoaset is retrieved. If an information item specified
23 by a query is already mapped in the partial infoaset, the infoaset model is not
24 augmented.

1 If, however, the information item specified by the query has not been
2 previously mapped, the partial info set is augmented by further mapping until the
3 specified information item is mapped. If no further mapping of the object is
4 required, the augmented info set model is stored for future reference.

5 Exemplary System

6 Fig. 4 is a block diagram depicting a system 400 suitable for use with the
7 systems and methods described herein. The elements shown in Fig. 4 and
8 described in relation thereto may be implemented as hardware components,
9 software modules or a combination of the two. Attribution of particular functions
10 to particular elements herein is not intended to limit performance of the functions
11 to the elements. Rather, any function included in the following description or any
12 portion thereof may be performed by any element or a combination of more than
13 one element.

14 The system 400 includes a processor 402, one or more input/output (I/O)
15 modules 404 and miscellaneous other hardware typically found in computing
16 systems that may be required to support the functionality described herein. One or
17 more of the I/O modules 404 may be a component configured to receive and/or
18 transmit electronic data, such as a network interface card. One or more of the I/O
19 modules may also be a component to facilitate entry of data or data processing
20 information, such as a keyboard or a pointing device.

21 The system 400 also includes memory 408 which may be random access
22 memory (RAM), flash memory, read only memory (ROM), cache memory or the
23 like. Although shown as a single element, it is noted that the memory 408 may be
24 a combination of any type of memory known in the art.

1 The memory 408 stores a filter engine 410, an operating system 412 and
2 miscellaneous software 414 that may be required to facilitate general system
3 functionality required to implement the techniques described herein.

4 The filter engine 410 includes a virtual machine 415, an input module 416
5 configured to receive an input message or some other form of electronic data
6 transmission and one or more filters 418 (queries) that may be stored as a filter
7 table or a plurality of filter tables. The filter engine 410 also includes a compiler
8 420 configured to parse a query and compile it into opcodes (operation codes)
9 needed to process the query, and an opcode store 422.

10 The filter engine 410 further includes a mapping module 424 that is
11 configured to map an object to an object info set (i.e. an object mapping) in an
12 info set store 426. One or more node sets 428 used for temporary storage during an
13 inverse query processing operation are associated with the filter engine 410.
14 Matching nodes obtained during a pass of a matching process are stored in the
15 node sets 428 so they can be used in subsequent passes of the matching process.

16 Additionally, the filter engine 410 includes a query processor 430 and a
17 query evaluator 432 that are configured to handle the matching process between
18 input values and filter values. An opcode execution module 434 is configured to
19 execute opcodes generated by the compiler 420 and a value retriever 436 is
20 configured to make a method call on an object to retrieve one or more values from
21 the object.

22 It is noted that although several elements are shown as being included
23 within the filter engine 410, one or more of these elements may be included in the
24 memory 408 or in one or more other components of the system 400. For
25

1 discussion purposes, such elements have been grouped with the filter engine 410
2 in the present example.

3 The system 400 is configured to communicate with a computing device 440
4 having a messaging module 442 that is configured to send and/or receive messages
5 to/from the system 400 over a network 444 such as the Internet.

6 Further reference will be made to the elements included in Fig. 4 in the
7 following discussion, and the elements contained therein will be discussed in
8 greater detail.

9 **Exemplary Methodological Implementation**

10 Fig. 5 is a flow diagram 500 depicting an exemplary methodological
11 implementation of evaluating a query formed according to a query language
12 format against an object structured according to a different format. In the
13 following discussion, the query is an XPath query and the object is a CLR object,
14 although it is noted that the query or the object may be structured according to one
15 or more other formats. In the following discussion, continuing reference will be
16 made to the elements and reference numerals shown in Fig. 4.

17 At block 502, the filter engine 410 identifies a CLR object context. If the
18 context is empty or there is nothing in the context to process (“Yes” branch, block
19 504), then process terminates. Otherwise (“No” branch, block 504), it is
20 determined if there is another level to evaluate in the query being evaluated
21 against the object. If there is not another level to evaluate, the process terminates.

22 If there is another level to evaluate (“Yes” branch, block 508) and there are
23 more objects in the context (“Yes” branch, block 510), a procedure to map an
24 appropriate info set model is called at block 512. If, for example, “Book/Title” has
25

1 been mapped but does not provide a match, now map, e.g., "Book/Author" if there
2 are such objects in the context.

3 It is noted that since the mapping cannot be accomplished until runtime,
4 certain obstacles must be overcome to complete the mapping. In a .NET
5 implementation, a technology called ".NET Reflection" is used to identify – or
6 "reflect" – a layout of an object at runtime, and can identify object properties,
7 methods, parameters, etc. The info set model can then be built with this
8 information.

9 At block 530, the object related to an info set being sought is identified, or
10 received. The object type is obtained (block 532) by, for example, using a
11 GetType method. If an appropriate info set model has already been created and is
12 cached ("Yes" branch, block 534), then the process returns to block 514. If
13 additional construction of the info set model is required ("No" branch, block 534),
14 the Type is reflected at block 536 and the info set model is built or augmented at
15 block 538. At this point, only the present Type is modeled in the info set. In other
16 words, no descendants of the Type are modeled at this point because they may not
17 be required. The info set model is cached at block 540 and the process reverts to
18 block 514 for subsequent processing.

19 At block 514, a particular information item is sought in the info set model.
20 In other words, has the mapping that has occurred to date mapped an information
21 item that is required to evaluate the query. If not ("No" branch, block 514), the
22 process reverts to block 506 to determine if there is another level in the query.

23 If the info set item includes the information items ("Yes" branch, block
24 514), a property or field matching the information item is selected at block 516.

1 This step pulls data associated with the information item. The matching data is
2 then added to a results cache, i.e. a nodeset, at block 518.

3 The results cache is made to be a current context at block 520 and the
4 process begins again at block 506 so that the new context (i.e. the previous results)
5 is processed.

6 Pseudo Code Algorithm

7 To help explain the procedure described above, the following pseudo code
8 example is provided. Assume that a query contains location path /a/b/c. The
9 location path is being tested against some object o. The location path has 3 steps.

10 Algorithm (pseudo code):

```
11 void Evaluate(LocationPath path, object input)
12 {
13     sourceNodeset.Add(input); // set up the initial nodeset
14     foreach (step in locationpath)
15     {
16         resultNodeset = PerformStep(step, sourceNodeset);
17         if (null == resultNodeset)
18         {
19             break; // nothing selected. Stop evaluating this
20                     // location path...
21         }
22         // perform the next step on the results of the previous step
23         sourceNodeset = resultNodeset;
24     }
25 }
```

```
19 Nodeset PerformStep(Step step, NodeSet nodeset)
20 {
21     foreach (obj in nodeset)
22     {
23         i. Retrieve the Type of obj by calling obj.GetType()
24         ii. If necessary, build a model for that Type and that Type
25             alone for but not of its descendants. i.e. there is no need
26             to map anything beyond 'level'1 yet. Only map obj. If obj
27             was a book, only map Book - don't look at chapters etc yet.
28             Any looks at the 'next level' will be driven by the
29             location path that is calling PerformStep. One step, one
30             level.
```



```
1      iii.  If the model does not contain an information item that maps
2            to 'step', then return null (nothing selected)
3      iv.   Else, perform the Select 'step' and place the resulting
4            nodes in a resultNodeset.
5      }
6      return resultNodeset;
7  }
```

Opcode Generation

8 In at least one implementation, the filter engine 410 compiler 420 generates
9 opcodes and maintains the opcodes in the opcode store 426 for future use.

10 Opcodes are instructions that are executed to evaluate a query. In the present
11 discussion, opcodes are used to execute the steps shown and described in Fig. 5.
12 The opcodes are very efficient because they dynamically generate 'helper' code to
13 perform the same function that would be performed by intermediate language
14 instructions compiled from source code. Generating code implies emitting MSIL
15 instructions into dynamic assemblies. This generated IL is just in time compiled by
16 the .NET runtime and allows the opcode to retrieve property values and fields
17 directly from CLR objects. It is as though the opcodes were executing compiled
18 hand-written source code.

19 Code is generated when an object of a particular Type/Class (Type/Class
20 are synonymous) or certain elements of the object are encountered for the first
21 time. Thereafter, when the same object elements are required to process a query,
22 the opcodes execute the already generated code to obtain the appropriate
23 information without having to perform more expensive procedures.

24 Although other methods may be utilized, the opcode generation process
25 increases efficiency of the systems and methods described herein. The following

1 discussion of opcodes is made with reference to the infoSet model 300 shown in
2 Fig. 3a, and the corresponding CLR Envelope Object shown above (reproduced
3 below for convenience).

```
4     public class Envelope
5     {
6         public ActionHeaderObj ActionHeader {get;}
7         public string Version {get;}
8     }
9
10    public class ActionHeaderObj
11    {
12        {XmlElement ("Action")}
13        public string ActionUri {get;}
14
15        {XmlElement ("HeaderName")}
16        public string Name {get;}
17
18        {XmlAttribute}
19        public string ID {get;}
20    }
```

13 Before the filter engine 410 can execute the location path
14 **ActionHeader/HeaderName** over an Envelope object, it compiles the XPath into
15 an opcode block containing one or more opcodes that can be executed within the
16 filter engine (or other type of virtual machine)

17 For the location path **ActionHeader/HeaderName**, the filter engine 410
18 opcode module 422 would produce, among others, the following *select* opcodes:

19 Select #1: to select an XML element named ActionHeader that is a child of
20 Envelope; and
21 Select #2: to select an XML element named HeaderName that is a child of
22 ActionHeader.

23 The first time that the filter engine 410 evaluates the select opcodes against
24 objects of a particular Type, it takes several preparatory steps. First - for Select #1
25 - the mapping module 428 constructs (or retrieves) at least a portion of an infoSet

1 model for the CLR-Type Envelope that was encountered. The filter engine 410
2 then looks for a child XML element named "ActionHeader" in the info set model
3 which, in the present example, is the ActionHeader property. If a child element
4 named "ActionHeader" does not already exist, the mapping module 428 creates a
5 wrapper CLR-Type for Envelope. Into each wrapper type, the mapping module
6 428 writes wrapper methods containing Intermediate Language (IL) to invoke
7 each public property in Envelope that is serializable into XML. This IL is
8 identical to what a compiler for a typical .NET language such as C# would
9 produce. For example, the wrapper type, if represented in the C#, would look like:

```
10     public class Envelope_Wrapper
11     {
12         public static ActionHeaderObj GetActionHeader(Envelope env)
13         {
14             return env.ActionHeader;
15         }
16         public static string GetVersion(Envelope env)
17         {
18             return env.Version;
19         }
20     }
```

17 The wrapper type is a class called Envelope_Wrapper. It contains an
18 equivalent static method for each property on the Envelope object. It could
19 contain similar methods for fields. Each method is written to retrieve the said
20 property from an instance of Envelope. Example, the wrapper method:

```
21     public static ActionHeaderObj GetActionHeader(Envelope env)
22     {
23         return env.ActionHeader;
24     }
```

24 retrieves the ActionHeader property from an object of type Envelope.
25

1 This is just one approach. There are other ways to generate equivalents,
2 including generating global methods or functions that are not necessarily
3 contained in a 'wrapping Type'.

4 It is noted that IL used to dynamically generate Envelope_Wrapper is
5 identical to that which the C# compiler would generate.

6 Since Select#1 opcode is attempting to retrieve an ActionHeader from
7 objects of Type Envelope, it must now call the GetActionHeader() method on
8 Envelope_Wrapper. This call, if represented in C#, would look like.

9 Envelope_Wrapper.GetActionHeader(obj);
10

11 The mapping module in 428 generates the IL for this call by creating a
12 delegate for the method GetActionHeaderObj. This delegate is called a selector.
13 It then saves this delegate inside opcode "Select #1."

14 The filter engine 410 then performs similar tasks related to Select #2. The
15 mapping module 428 attempts to locate an infoSet model for the ActionHeaderObj
16 CLR-Type. If no infoSet model can be located, at least a portion of such an infoSet
17 model is created (see flow diagram, 500, Fig. 5). The mapping module 428 then
18 attempts to locate a child XML element names HeaderName in the infoSet model
19 which, in the present example, is the Name property.

20 If such a CLR-Type does not already exist, the filter engine 410 generates a
21 wrapper CLR-Type for ActionHeaderObj. Into each wrapper type, it writes
22 wrapper methods containing IL to call each public property in ActionHeaderObj
23 that is serializable into XML. A selector is created to invoke the wrapper method
24 for the Name property and the selector is saved into "Select #2."
25

1 Although not shown, the wrapper types are maintained by the filter engine
2 410 in a dynamic assembly (in a .NET implementation). The dynamic assembly is
3 an assembly code module that is generated dynamically (and just in time compiled
4 by the .NET runtime). The wrapper CLR Types and delegates are included in this
5 dynamic assembly.

6 It is noted that not all selectors need be generated automatically. The filter
7 engine 410 can also support custom hand-coded selectors designed and optimized
8 to work with specific CLR-Types to further enhance efficiency of particular
9 systems.

10 Finally, to execute the location path **ActionHeader/HeaderName** over an
11 instance of the Envelope class, the filter engine 410 first evaluates “Select #1” by
12 invoking its selector, which selects the ActionHeader property from Envelope and
13 places the result in the nodeset 434. The filter engine 410 then evaluates “Select
14 #2” by invoking its selector for each node in the nodeset 434 from the previous
15 step.

16 Subsequent query evaluations that require execution of the
17 **ActionHeader/HeaderName** location path will execute the opcodes (i.e. “Select
18 #1” and “Select #2”). Execution of these opcodes is very efficient and
19 significantly reduces system resources that are required to evaluate such queries.
20

21 **Exemplary Computer Environment**

22 The various components and functionality described herein are
23 implemented with a computing system. Fig. 6 shows components of typical
24 example of such a computing system, i.e. a computer, referred by to reference
25 numeral 600. The components shown in Fig. 6 are only examples, and are not

1 intended to suggest any limitation as to the scope of the functionality of the
2 invention. Furthermore, the invention is not necessarily dependent on the features
3 shown in Fig. 6.

4 Generally, various different general purpose or special purpose computing
5 system configurations can be used. Examples of well known computing systems,
6 environments, and/or configurations that may be suitable for use with the
7 invention include, but are not limited to, personal computers, server computers,
8 hand-held or laptop devices, multiprocessor systems, microprocessor-based
9 systems, set top boxes, programmable consumer electronics, network PCs,
10 minicomputers, mainframe computers, distributed computing environments that
11 include any of the above systems or devices, and the like.

12 The functionality of the computers is embodied in many cases by computer-
13 executable instructions, such as program modules, that are executed by the
14 computers. Generally, program modules include routines, programs, objects,
15 components, data structures, etc. that perform particular tasks or implement
16 particular abstract data types. Tasks might also be performed by remote
17 processing devices that are linked through a communications network. In a
18 distributed computing environment, program modules may be located in both local
19 and remote computer storage media.

20 The instructions and/or program modules are stored at different times in the
21 various computer-readable media that are either part of the computer or that can be
22 read by the computer. Programs are typically distributed, for example, on floppy
23 disks, CD-ROMs, DVD, or some form of communication media such as a
24 modulated signal. From there, they are installed or loaded into the secondary
25 memory of a computer. At execution, they are loaded at least partially into the

1 computer's primary electronic memory. The invention described herein includes
2 these and other various types of computer-readable media when such media
3 contain instructions programs, and/or modules for implementing the steps
4 described below in conjunction with a microprocessor or other data processors.
5 The invention also includes the computer itself when programmed according to
6 the methods and techniques described below.

7 For purposes of illustration, programs and other executable program
8 components such as the operating system are illustrated herein as discrete blocks,
9 although it is recognized that such programs and components reside at various
10 times in different storage components of the computer, and are executed by the
11 data processor(s) of the computer.

12 With reference to Fig. 6, the components of computer 600 may include, but
13 are not limited to, a processing unit 602, a system memory 604, and a system bus
14 606 that couples various system components including the system memory to the
15 processing unit 602. The system bus 606 may be any of several types of bus
16 structures including a memory bus or memory controller, a peripheral bus, and a
17 local bus using any of a variety of bus architectures. By way of example, and not
18 limitation, such architectures include Industry Standard Architecture (ISA) bus,
19 Micro Channel Architecture (MCA) bus, Enhanced ISA (EISAA) bus, Video
20 Electronics Standards Association (VESA) local bus, and Peripheral Component
21 Interconnect (PCI) bus also known as the Mezzanine bus.

22 Computer 600 typically includes a variety of computer-readable media.
23 Computer-readable media can be any available media that can be accessed by
24 computer 600 and includes both volatile and nonvolatile media, removable and
25 non-removable media. By way of example, and not limitation, computer-readable

1 media may comprise computer storage media and communication media.
2 “Computer storage media” includes volatile and nonvolatile, removable and non-
3 removable media implemented in any method or technology for storage of
4 information such as computer-readable instructions, data structures, program
5 modules, or other data. Computer storage media includes, but is not limited to,
6 RAM, ROM, EEPROM, flash memory or other memory technology, CD-ROM,
7 digital versatile disks (DVD) or other optical disk storage, magnetic cassettes,
8 magnetic tape, magnetic disk storage or other magnetic storage devices, or any
9 other medium which can be used to store the desired information and which can be
10 accessed by computer 600. Communication media typically embodies computer-
11 readable instructions, data structures, program modules or other data in a
12 modulated data signal such as a carrier wave or other transport mechanism and
13 includes any information delivery media. The term “modulated data signal”
14 means a signal that has one or more of its characteristics set or changed in such a
15 manner as to encode information in the signal. By way of example, and not
16 limitation, communication media includes wired media such as a wired network or
17 direct-wired connection and wireless media such as acoustic, RF, infrared and
18 other wireless media. Combinations of any of the above should also be included
19 within the scope of computer readable media.

20 The system memory 604 includes computer storage media in the form of
21 volatile and/or nonvolatile memory such as read only memory (ROM) 608 and
22 random access memory (RAM) 610. A basic input/output system 612 (BIOS),
23 containing the basic routines that help to transfer information between elements
24 within computer 600, such as during start-up, is typically stored in ROM 608.
25 RAM 610 typically contains data and/or program modules that are immediately

1 accessible to and/or presently being operated on by processing unit 602. By way
2 of example, and not limitation, Fig. 6 illustrates operating system 614, application
3 programs 616, other program modules 618, and program data 620.

4 The computer 600 may also include other removable/non-removable,
5 volatile/nonvolatile computer storage media. By way of example only, Fig. 6
6 illustrates a hard disk drive 622 that reads from or writes to non-removable,
7 nonvolatile magnetic media, a magnetic disk drive 624 that reads from or writes to
8 a removable, nonvolatile magnetic disk 626, and an optical disk drive 628 that
9 reads from or writes to a removable, nonvolatile optical disk 630 such as a CD
10 ROM or other optical media. Other removable/non-removable,
11 volatile/nonvolatile computer storage media that can be used in the exemplary
12 operating environment include, but are not limited to, magnetic tape cassettes,
13 flash memory cards, digital versatile disks, digital video tape, solid state RAM,
14 solid state ROM, and the like. The hard disk drive 622 is typically connected to
15 the system bus 606 through a non-removable memory interface such as data media
16 interface 632, and magnetic disk drive 624 and optical disk drive 628 are typically
17 connected to the system bus 606 by a removable memory interface such as
18 interface 634.

19 The drives and their associated computer storage media discussed above
20 and illustrated in Fig. 6 provide storage of computer-readable instructions, data
21 structures, program modules, and other data for computer 600. In Fig. 6, for
22 example, hard disk drive 622 is illustrated as storing operating system 615,
23 application programs 617, other program modules 619, and program data 621.
24 Note that these components can either be the same as or different from operating
25 system 614, application programs 616, other program modules 618, and program

1 data 620. Operating system 615, application programs 617, other program
2 modules 619, and program data 621 are given different numbers here to illustrate
3 that, at a minimum, they are different copies. A user may enter commands and
4 information into the computer 600 through input devices such as a keyboard 636
5 and pointing device 638, commonly referred to as a mouse, trackball, or touch
6 pad. Other input devices (not shown) may include a microphone, joystick, game
7 pad, satellite dish, scanner, or the like. These and other input devices are often
8 connected to the processing unit 602 through an input/output (I/O) interface 640
9 that is coupled to the system bus, but may be connected by other interface and bus
10 structures, such as a parallel port, game port, or a universal serial bus (USB). A
11 monitor 642 or other type of display device is also connected to the system bus
12 606 via an interface, such as a video adapter 644. In addition to the monitor 642,
13 computers may also include other peripheral output devices 646 (e.g., speakers)
14 and one or more printers 648, which may be connected through the I/O interface
15 640.

16 The computer may operate in a networked environment using logical
17 connections to one or more remote computers, such as a remote computing device
18 650. The remote computing device 650 may be a personal computer, a server, a
19 router, a network PC, a peer device or other common network node, and typically
20 includes many or all of the elements described above relative to computer 600.
21 The logical connections depicted in Fig. 6 include a local area network (LAN) 652
22 and a wide area network (WAN) 654. Although the WAN 654 shown in Fig. 6 is
23 the Internet, the WAN 654 may also include other networks. Such networking
24 environments are commonplace in offices, enterprise-wide computer networks,
25 intranets, and the like.

1 When used in a LAN networking environment, the computer 600 is
2 connected to the LAN 652 through a network interface or adapter 656. When used
3 in a WAN networking environment, the computer 600 typically includes a modem
4 658 or other means for establishing communications over the Internet 654. The
5 modem 658, which may be internal or external, may be connected to the system
6 bus 606 via the I/O interface 640, or other appropriate mechanism. In a networked
7 environment, program modules depicted relative to the computer 600, or portions
8 thereof, may be stored in the remote computing device 650. By way of example,
9 and not limitation, Fig. 6 illustrates remote application programs 660 as residing
10 on remote computing device 650. It will be appreciated that the network
11 connections shown are exemplary and other means of establishing a
12 communications link between the computers may be used.

Conclusion

The systems and methods as described thus provide an efficient way to test queries structured according to a first protocol (such as XML queries) against inputs containing objects structured according to a second protocol that is different from the first protocol (such as CLR objects). Since an object does not have to be serialized, the costly serialization step is not executed, thereby reducing system overhead required to evaluate queries against the object.

Although details of specific implementations and embodiments are described above, such details are intended to satisfy statutory disclosure obligations rather than to limit the scope of the following claims. Thus, the invention as defined by the claims is not limited to the specific features described above. Rather, the invention is claimed in any of its forms or modifications that fall within the proper scope of the appended claims, appropriately interpreted in accordance with the doctrine of equivalents.